



Design Tips for Better Performance White Paper

VERSION 2022

Jade Software Corporation Limited cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages, or loss of profits. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Jade Software Corporation Limited.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Copyright © 2025 Jade Software Corporation Limited.

All rights reserved.

JADE is a trademark of Jade Software Corporation Limited. All trade names referenced are the service mark, trademark, or registered trademark of the respective manufacturer.

For details about other licensing agreements for third-party products, you must read the Jade **Readme.txt** file.

Contents

Contents	iii
Design Tips for Better Performance	4
What Changed with Jade 7.0?	4
Overview	4
Designing for Performance and Scalability	5
Micro-Benchmarking - Measure as You Go	5
Monitoring Performance	6
Improving Performance	7
Jade System Installation and Configuration	8
Processing Performance - Design Issues	12
Method Cache	12
Caching Strategy	12
Exception Handling Strategy	12
Improving the Efficiency of Deletes	13
Processing Performance - Development Issues	13
Iterating through Arrays	14
serverExecution Method Option	14
Deleting Transients	15
Jade Methods versus External Methods and Functions	16
Very Large Methods, with Most Code Unused	16
Method Calls	16
Fetching Properties	17
Excessive Logging	17
Windows Calls – Setting Table Cell Text	17
Multiple Concatenations	18
System Methods or System Properties?	18
beginTransaction and commitTransaction Instructions	19
IO Performance – Design Issues	20
Slobs and Blobs	20
Avoiding firstInstance and instances Methods	21
Locking Strategy	21
Using Local Collections	22
Setting Key Values before References	22
Map Files	22
IO Performance – Development Issues	23
Iterator versus the foreach Instruction	24
Usage	24
Locking Behavior	24
Starting Part-Way through the Collection	25
Using Collection Keys to Avoid Fetching Objects	26
Minimizing the Working Set	26
Jade Implicit Locking	27
Releasing Locks	27
Large Batch Processes (Adds, Deletes, or Updates)	28
Reducing Deadlocks	29
Defining a Locking Strategy	30

Design Tips for Better Performance

This document contains specific design and coding hints and tips for Jade developers. It is organized by area rather than in order of importance.

For guidance in analyzing the performance of a Jade system, refer to the [Performance Analysis](#) white paper.

This is not an exhaustive discussion of the subject, and is intended to provide understanding and guidance, and perhaps encourage the motivation to research other techniques as the need arises.

There is a lot of information in this white paper. It may pay to come back in a few months' time and peruse it again. Other things will likely jump out at you then. You should read this along with Part 4, "Design Considerations", in the [Erewhon Demonstration System Reference](#) that is also available from the **JADE-Erewhon** link at <https://github.com/jadesoftwarez>.

Note This white paper is relevant to Jade 7.0 and higher, including Jade 2022. In earlier releases, a number of mechanisms operated differently, and it is likely that many may change with future releases. In particular, Jade 7.0 and higher use their own disk cache as the primary cache for database objects, while Jade 6.3 and earlier use Windows file-system cache. In addition, Jade 7.0 and higher support much better concurrency between multiple processes within a node. The comparison with Jade 6.3 is still useful because many systems were initially configured for Jade 6.3 or earlier, and have not been updated since.

For more details, see the following subsections.

What Changed with Jade 7.0?

Since Jade 7.0, the persistent database implements its own disk cache. This allows for more flexibility and performance optimization than was available with Windows file-system cache.

Much-greater concurrency is possible for processes within a node. The protocols for sharing common resources - including the Jade Object Manager (JOM) persistent object cache - have been improved to provide better performance and support more processes per node.

The benefits of 64-bit Jade are discussed in the "[Jade System Installation and Configuration](#)" section of this document. With Jade 7.0 and higher, the database server must be 64-bit. Application server and fat client nodes should normally be 64-bit, but they *can* be 32-bit, if necessary.

There is a new RPC transport (**HPSM**), which allows for greater concurrency than the **JadeLocal** transport.

Overview

Most performance problems are caused by excessive consumption of finite resources. Performance improvement can be seen as a process of improving the management of scarce resources.

Identifying your most-scarce resources will help you focus your optimizations where they can do the most good. As you optimize the use of one resource, another resource may become your most scarce. Sometimes there are direct trade-offs between resources. Using resources is not a problem, in fact it is necessary, but wasting them or balancing them poorly results in unnecessary performance problems.

For more details, see the following subsections.

Designing for Performance and Scalability

Many performance problems surface when system workload is increased; for example, when the database grows or transaction volumes increase.

If the system is scalable, performance levels will be sustained at acceptable levels as the workload increases. If it is not scalable, performance may drop considerably as the workload increases.

For best results, design for performance and scalability up front.

Even when you do that, though, you may still encounter increased workload in unexpected areas. In those cases, general system design characteristics like modularization through layering are very helpful for diagnosis and for the implementation of the required modifications.

Micro-Benchmarking - Measure as You Go

While designing or coding, you may be aware of different ways of accomplishing the same functionality in Jade. It can be beneficial to do a performance test to decide between them at the time, particularly if the same functionality will be used many times. Simple construct choices, in particular, benefit from measuring as you go, as they are easier to test.

Micro-benchmarking refers to a simple test comparing two constructs, to see which is faster. These tests are frequently run in Workspace or **JadeScript** methods. The following is an example **JadeScript** method that is useful for comparing constructs.

```
jblPfm() updating;
vars
  oldWay : Boolean;
  dts,dte : Decimal[19];
  s,t,u,v : String;
  i,j, reps : Integer;
begin
  write CrLf & "Starting...";
  reps := 10000; // adjust this to get a run length of 1 to 10 seconds
  oldWay := true;
  dts := app.relativeMachineTime;
  foreach i in 1 to reps do
    if oldWay then
      s := app.actualTimeServer.String;
    else
      s := app.actualTimeAppServer.String;
    endif;
  endforeach;
  dte := app.relativeMachineTime;
  write "<" & s & ">" & "<" & t & ">"; // to verify the functional results
  write ((dte-dts)/reps).roundedTo(6).String & " ms/rep";
epilog
  write "Finished";
end;
```

When performing micro-benchmarks, there are a few things to watch out for.

- If the test involves reading objects from the database, be aware that the objects will likely be in cache the second time the test is run. It is all too easy to test something the old way, make a change, run it again, see a big improvement, and assume it was the change that made the difference.

It is possible that the primary difference was the fact that objects were cached the second time around - in the node's cache or the database server's disk cache.

If you need a reliable comparison of tests involving database IO, you will need to restart the database server between tests, to clear the database server's disk cache. This may be a big inconvenience, but it is essential for tests involving IO.

- If you measure your optimizations in a hot cache scenario (by not restarting), small differences in CPU can seem significant, when in real life they would be insignificant compared to the IO that is being done.
- If you are analyzing results using the Jade Monitor, be aware that running a Workspace or **JadeScript** method involves starting and stopping a Jade application, which may have a significant impact on the counters you are examining.

Run an empty **JadeScript** method and analyze the activity involved, to see what the overhead is. The overheads can be especially large if the test uses classes that no other processes in the node are using, which is common on development systems. Unless the classes are already in use or Production mode is being used, executing a **JadeScript** method will incur the overhead of opening the classes. In addition, when the **JadeScript** method finishes, the classes will be closed (unless used elsewhere or Production mode is on), and all instances of those classes will be removed from the node's cache. This also applies to Workspaces. This can lead to big inconsistencies between runs; for example, if another user has the classes open for some runs but not other runs.

As an alternative, you may want to use a simple form that allows you to run a test by clicking a button and displaying the results in a large text box display. This way you can avoid the overheads of starting and stopping an application, and the potential variability of opening and closing classes.

If you *do* use a **JadeScript** or Workspace method, be sure to measure the time internally (as in the previous code example), and create and delete any work transients and so on outside the timed section – unless that is what you are benchmarking, of course!

Monitoring Performance

It is highly recommended that you use the **JadeMonitor** application regularly, to monitor the performance characteristics of your system, particularly after making major changes to an area of code. You should check the following things in the **JadeMonitor** application while unit testing a specific function; for example, a new button click.

- Use the **Method Analysis** sheet for profiling. Investigate the methods with the most Clock Time, Clock Time Method, and CPU Time Method. Concentrate on the top of the lists. If it's not in the top 10 percent or 20 percent, it's probably not important.

Checking the number of calls to methods can help you spot application logic errors. For example, should that date routine be called for each child object or each parent object?

- Use the **Process Information** sheets for function counts (register for remote *and* local).
 - Were you expecting 50,000 persistent Get Objects for this button click? 100,000 locks?
 - Were you expecting 8,000 transient object creates and only 1,000 deletes?
 - Is this a transient leak?

If you have the system to yourself, you can use System Statistics or Node Statistics for a more-concise list of the more-important counters.

- If you see some counts or times larger than you expected, you might be able to use the **Persistent Object Activity** sheets to track down the classes involved.

If the number of locks is high, use the locks summary and detail sheets to identify what is being locked. Keep in mind that these sheets monitor *all* users of the database, so they are more useful when there are no other active users.

To capture performance information for specific actions (for example, a button click), refresh (F5) the sheet before and after the button click. Do not use the timer for this.

If there are multiple sheets of interest, use the **Monitor Setup** sheet to select all sheets of interest for periodic sampling. This means you can use Ctrl+F5 to refresh all of the panes simultaneously (before and after the button click), giving you a coherent snapshot of the activity caused by the button click. Don't select all sheets for periodic sampling unless you really need it.

Always log everything to file, though (by right-clicking and then selecting the **Select All** item from the popup menu) so that everything you see on the monitor form as well as the periodically sampled items are logged. You never know if you will want to refer back to it later, or send the log file to a colleague. If you want to have a log file containing a specific test, release the log file before and after the test.

Note that some sheets have a **Begin Monitoring** button, or similar. These are usually the sheets that have a more-significant impact on system performance while monitoring, so they should not generally be left monitoring for extended periods.

You should check the following things while integration testing or load testing a new release of an application prior to implementing it in the production system. Generally, these should be used with a timer (usually 10 or 30 seconds, depending on the length of the run) and the results logged to a file. Use a second **JadeMonitor** application for the ad hoc inquiries.

- On the **Host Performance** sheet **Summary** metric, the most important counters are usually the queue depths on the database and logs drives and the total % **Processor**.
- The **System Statistics** sheet gives you a good high-level view of the number of Jade functions that are being performed.

It pays to get familiar with the Rate statistics (counts per second); for example, the number of Get Objects per second that are typical for your system under load.

- The **Node Statistics** sheet is useful for splitting out the high-level counters by node.
- Monitor queued locks on the timer for the entire run.

Use the **Persistent Object Activity** sheet for occasional short periods, to get an idea of what classes are used the most.

- Check locks, locks summaries and details, and lock contentions periodically, or if there appear to be locking issues. High rates of mutex contentions can indicate node contention.
- If there is a lot of IO activity on the database drive (monitored on the **Host Performance** sheet), the **DbFile Analysis** sheet may help identify the map files and therefore the classes involved.

Improving Performance

Excessive consumption of CPU or IO can stem from inefficient coding, inappropriate initialization file settings, or inadequate hardware. Excessive contention for locks can be caused by poor data model design, lack of a good locking strategy, or poor operational procedures.

Generally speaking, you improve performance by reducing the amount of work done. Specifically, you can look into the following areas.

- Ensure that database queries access only the required data.
- Minimize the number of trips to the database server to fetch or lock objects. This means ensuring the Jade Object Manager (JOM) persistent object cache sizes in Jade nodes are large enough, to avoid re-fetching the same objects. It also means having a good locking strategy.

It also helps to minimize the use of constructs like [isValidObject](#), [actualTimeServer](#), and [serverExecution](#), which require trips to the server.

- Minimize the number of trips to the disk subsystem. All you have to do is add memory and increase the value of the **DiskCacheMaxSegments** parameter in the [PersistentDb] section of the Jade initialization file to a value that takes advantage of the extra memory.

The default setting of the **DiskCacheMaxSegments** parameter is a good starting point if there is only one database on the machine. If you have multiple databases on one machine (for example, development and UAT systems), choose lower settings of **DiskCacheMaxSegments**, to allocate the memory amongst them.

For more details, see the [Server Memory Allocation](#) white paper.

- Minimize the number of messages between presentation clients and application servers. This requires care in coding and an awareness of the constructs that cause these messages to be sent.

Although this is not by any means a complete list, this white paper addresses the more-common issues that have been seen by Jade users.

Jade System Installation and Configuration

The installation and configuration options you specify for an application can have a significant impact on overall performance.

Some of the options available to you are as follows.

- Thin client

In Jade thin client mode, the presentation clients are serviced by an application server (effectively a special type of fat, or standard, client), which performs processing on their behalf and transfers over the network only the minimum information required to support the GUI on the presentation client. The application server would typically be located with the database server, allowing a high performance connection between them. Thin client is generally the best option for interactive users when deploying over any network.

Sending large messages over the network can slow down thin client processing; for example, large graphics or repeated updates of GUI elements. Some GUI constructs can cause a lot of network traffic; for example, the [TextBox::change](#) and [Form::keyDown](#) event methods. See "[Jade Thin Client Performance Considerations](#)" in Appendix A of the *Thin Client Guide*.

The thin client trace facility can help analyze thin client network traffic.

- Web browser

When running HTML thin client mode using the Internet to access applications, Jade generates HTML-based forms that are accessed using a web browser. This is similar in some respects to the Jade thin client, except that you are constrained to some degree by the limitations and functionality of HTML and the browsers themselves.

- Fat client

In fat (standard) client mode, each Jade client performs full Jade node processing (Jade method execution, object caches, and so on) and presentation client functions such as printing. Fat clients should normally be located with the database server, with a high-performance connection between them; for example, the **HPSM** or **JadeLocal** transport. Background processing applications would normally be run as fat clients.

Performance is typically reduced when deploying over a Local Area Network (LAN) or Wide Area Network (WAN), as the volume of network traffic and network speed impact on performance. See also Distributed Processing, later in this section.

- Server applications

Server applications run on the database server node. These would normally be system control-type applications. Database access is faster from the server node, but there are restrictions. For details, see the discussion about the **serverExecution** method option, later in this section.

GUI functions, including printing, are not available on the server node.

It may be easier to stop and restart applications if they are running as fat clients rather than server applications.

- **JadeLocal** transport

The **JadeLocal** transport uses shared memory so it is significantly faster than TCP/IP for nodes located on the same machine as the database server. Application servers and fat client nodes therefore should normally be located on the same machine as the database server if possible, so that they can benefit from the use of **JadeLocal** or **HPSM** transport.

- **HPSM** transport

The Hybrid Pipe and Shared Memory (**HPSM**) high-performance transport was new in Jade 7.0. Like the **JadeLocal** transport, **HPSM** uses shared memory so it is significantly faster than TCP/IP for nodes located on the same machine as the database server.

The **HPSM** transport has lower overhead, fewer context switches, and higher concurrency than **JadeLocal** transport.

- 64-bit Jade

Starting with the 6.3 release, 64-bit binaries were available. With Jade 7.0 and later, the database server must be 64-bit. Application server and fat client nodes should normally be 64-bit, but they can be 32-bit if necessary; for example, to use a third-party DLL that is available only in 32-bit.

64-bit binaries have the following advantages over 32-bit binaries.

- Persistent and transient caches can be larger than 2G bytes. Real-world tests have confirmed that using a 3G byte persistent object cache in a node can improve its performance.

Contrived tests have confirmed that re-reading a large amount of data from the database was much quicker in a node with a 22G byte persistent object cache.

- The limit of 2G bytes of virtual memory per 32-bit process is effectively removed. This means that cache sizes can be set more aggressively without concern about exceeding the 2G byte limit.

This especially relates to method cache and string pool sizes, and to the number of server threads in the database server.

64-bit Jade provides the ability to address more memory and therefore allocate larger caches. It is important to consider the overall memory allocation strategy when considering the size of a cache. Selecting too large a cache size could result in excessive paging due to insufficient physical memory. As part of this strategy it is important to allow sufficient memory for PDB disk cache.

Note External DLLs must be compiled as 64-bit versions.

You can run an application server or fat client in 32-bit mode; for example, if a third-party DLL is not available in 64-bit.

- Reorganization fast collection build

You can specify in your initialization file that fast collection builds should be used. This option causes a reorganization to extract and sort keys to build a collection, rather than inserting them one at a time. For large collections, this can save a lot of random IO.

```
[JadeReorg]
FastBuildBTreeCollections=true
```

A test was done with three global collections that had 76 million entries combined. All three collections were reorganized by changing a key to descending. Without fast build, the reorganization took 98 minutes. With fast build, the same reorganization took 15 minutes.

For some restrictions about when this can be used, see Chapter 14, "[Database Reorganization](#)", in the *Developer's Reference*.

- Asynchronous method calls

Asynchronous method calls can be used to improve throughput by increasing concurrency.

A test was done with a number of CPU-bound tasks and varying numbers of asynchronous worker threads on a machine with 24 CPUs. The total throughput (in terms of calculations per minute) increased as the number of workers increased from 1 to 24. Using more than 24 workers did not provide further improvement, as there was not enough CPU power to support them.

Another test was done with a number of IO bound tasks. Each task consisted of 2,000 [getAtKeys](#) randomly distributed throughout several large global collections. The test was performed on a machine with a high-end Storage Area Network (SAN). The overall throughput (in terms of [getAtKeys](#) per second) increased as the number of workers increased, up to the point where several dozen workers were running.

One optimization used by high-end SANs is to maximize concurrency in a Raid 10 (mirrored and striped) configuration, by choosing which disk spindle to read from, with multiple operations occurring simultaneously. The same test was run on a machine with a more-basic disk controller and a smaller number of physical spindles. The overall throughput reached its peak with about 15 workers.

In both cases, the IO workers took longer to perform each task, but overall productivity increased due to the increased concurrency.

For details about asynchronous method calls, see the [Asynchronous Method Calls](#) white paper.

- Distributed processing

Distributed processing refers to locating Jade nodes (fat clients and application servers) on machines other than the database server machine. This can provide performance benefits for larger configurations.

Locating all nodes on the same physical machine and using the **HPSM** or **JadeLocal** transport generally provides the fastest communication path. However, if that machine is overloaded, overall performance may be improved by moving some fat clients or application servers to other machines that have fast network connections.

Any object retrieval, lock, update, notification, or other message with the database server uses the communications transport, so in general the inquiry nodes should be off-loaded to other machines and the nodes with heavy updating of the database should be kept local to the database server. This is because updating requires more round-trip messages between the node and the database server, and you want those trips to be as fast as possible.

Consider the following code fragment.

```
beginTransaction;  
    parent.data := dataIn;  
commitTransaction;
```

The execution of these three lines requires *seven* round trips from the Jade node to the database server. These can be measured using the **RPC Activity Summary** sheet in the **JadeMonitor** application. The trips are:

- beginTrans
- LockObject
- CommitTrans (commit processing is starting)
- PutObject
- endTrans (finalize the updates and make them available)
- remove Locks
- returnNotes (sends deferred notifications)

Each node that is moved to another machine increases the load on the database server, because TCP/IP requires more CPU than the **HPSM** or **JadeLocal** transport. Therefore, start by moving a small number of nodes whose function is mainly inquiry. Increasing the size of the persistent object caches in the nodes on remote machines may improve performance by reducing the number of objects fetched from the database server.

The use of the **serverExecution** method option is more attractive from remote nodes, as the cost of trips to the database server is greater. Don't overuse it, though, or you will end up with all processing on the database server machine again.

Scalability is not the only reason for using distributed processing; for example, some systems use a fat client node on an FTP server machine outside a firewall, connected to a database server on a machine inside the firewall.

- Single user application server

Where you need to have only one application server running, you could consider running in single user mode, which offers better performance over multiuser mode as it does not have the overheads associated with supporting multiple nodes.

Note that *single user* and *multiuser* refer to the number of nodes; not to the number of end-users. If there are too many end-users for a single application server or if background fat client nodes are required, single user mode is not suitable.

Processing Performance - Design Issues

This section provides processing performance design issue tips.

- [Method Cache](#)
- [Caching Strategy](#)
- [Exception Handling Strategy](#)
- [Improving the Efficiency of Deletes](#)

Method Cache

If multiple method caches are used, each Jade application has its own method cache and there is no contention.

If a single method cache is used, all applications in the node contend for that cache. Better performance therefore normally results from multiple caches, at the cost of additional memory.

Caching Strategy

Every multiuser Jade application requires a caching strategy. A good caching strategy ensures that the objects stored in your local cache are the latest editions, and that this is maintained with the minimum amount of network and processing activity.

An application that makes efficient use of cache will have significant performance advantages over one that does not. Jade's distributed cache capability is one of its key strengths.

In general, you should use Jade's automatic cache coherency. With automatic cache coherency enabled, objects updated in other nodes (database server, application servers, background nodes, or fat clients) are automatically reloaded in local cache if they are accessed again.

You can use edition checking in an optimistic strategy for detecting concurrent updates in a simple edit scenario. For more details, see "Edition Checking", in Part 4, "Design Considerations", of the *Erewhon Demonstration System Reference* included with the Example schemas and also available on the Jade website at <https://www.jadeplatform.com/developer-centre/learn/whitepapers>.

For details about designing and building an effective caching strategy, see "Cache Synchronization" in Part 4, "Design Considerations", of the *Erewhon Demonstration System Reference*.

Exception Handling Strategy

You should include a coherent exception handling strategy as part of your system design. The basis of your strategy should be an assumption that 99 percent of the time errors will not occur. Rather than perform checks for possible errors (such checks being a waste of CPU cycles, 99 percent of the time), use exception handlers to catch the 1 percent of transactions that produce errors. This is far more efficient, and makes your code easier to read and maintain. However, you *do* need to plan for this approach at the outset, to gain maximum benefit from it.

Be careful not to engage in "ostrich coding" - explicitly detecting and ignoring inconsistencies. These are normally characterized by excessive use of the `isValidObject` method and checks for null. For example, consider the following code.

```
if myParent <> null
and myParent.isKindOf(Parent)
and app.isValidObject(myParent)
```

```
and myParent.hasOutstandingBalance then
  // do something
endif;
```

If the data model does not allow for this object's **myParent** reference to validly be null, this code is detecting database errors and deliberately ignoring them. In addition, the **isValidObject** method does not ensure that the object will be valid when it is subsequently accessed. It is better to assume the data is correct, and let an exception handler take a stack dump if it is not, as shown in the following code fragment.

```
if myParent.hasOutstandingBalance then
  // do something
endif;
```

In addition to being easier to read, this code executes faster when the extraneous condition checks are removed. The compiled methods are also smaller, making more-effective use of method cache.

For more details about designing and building an effective exception handling strategy, you should read:

- The [Jade Exception Handling](#) white paper.
- "Exception Handling" in Part 4, "Design Considerations", in the *Erewhon Demonstration System Reference*.

Improving the Efficiency of Deletes

When defining inverse relationships, you have the option of defining the relationship as parent/child. This relationship means that if you delete the parent, Jade will automatically delete the children. This is more efficient than coding the deletes of the children yourself, as Jade does not have to process its code through the interpreter. However, all of the deletes will be done in a single transaction, so this may not be appropriate for large groups of objects.

When deleting children by using this parent/child relationship, you should use destructor methods on the child classes to manage the integrity of other relationships that exist with the child classes.

Processing Performance - Development Issues

This section provides processing performance development issue tips.

- [Iterating through Arrays](#)
- [serverExecution Method Option](#)
- [Deleting Transients](#)
- [Jade Methods versus External Methods and Functions](#)
- [Very Large Methods, with Most Code Unused](#)
- [Method Calls](#)
- [Fetching Properties](#)
- [Excessive Logging](#)
- [Windows Calls - Setting Table Cell Text](#)
- [Multiple Concatenations](#)

- [System Methods or System Properties](#)
- [beginTransaction and commitTransaction Instructions](#)

Iterating through Arrays

The most-efficient mechanism for iterating through an array is to use a **foreach** instruction loop; for example, consider the following code fragment.

```
vars
    nameArray : StringArray;
    name : String;
    i,nameCount : Integer;
begin
    nameCount := nameArray.size;
    i := 1;
    while i < nameCount do
        name := nameArray[i];
        // do something with name
        i := i+1;
    endwhile;
```

This code is inefficient because arrays are implemented as lists, so **nameArray[i]** has to start at the beginning of the list and count through to find the item. In one test, **objectArray[20000]** took nine times as long as **objectArray[1]**. If there are millions of entries in the array, the performance of this construct is even worse. Assuming **nameArray** is persistent, the fact that **nameArray.size** and **nameArray[i]** lock and unlock the array is another good reason not to call them repeatedly.

This can be more efficiently performed by the following code.

```
vars
    nameArray : StringArray;
    name : String;
begin
    foreach name in nameArray do
        // do something with name
    endforeach;
```

This was tested with an array containing 20,000 names. The **while** loop took 1.6 seconds and the **foreach** loop took 0.046 seconds.

serverExecution Method Option

The **serverExecution** method option historically was used a lot, prior to the introduction of Jade thin client. Jade nodes (fat clients) were on people's workstations and it was indeed a long way to go over the network to the database server. Saving trips to the server was much more important back then, and saving trips is primarily what this method option is used for.

Nowadays, most end-users are on thin clients and most nodes are on the same physical machine as the database server, using the **HPSM** or **JadeLocal** transport (shared memory). Saving trips to the server is not nearly as important now. However, it still can be important.

In Jade 6, excessive use of **serverExecution** could result in node contention, especially if large numbers of objects were being fetched from the database. With Jade 7 and higher, that node contention is effectively removed, meaning that you can feel more free to use **serverExecution**.

The **serverExecution** method option is useful in the following situations.

- A lot of database objects need to be retrieved and very little information is returned.
- A lot of updating is done, with little code execution in between; for example, cloning a number of persistent objects.
- A centralized location is required; for example, keeping track of the next global unique number to be used.

However, be aware that there are some pitfalls, including the following.

- The method is executed on a server long thread. If you increase the amount of server executions, you may need to increase the value of the **MaxLongThreads** parameter in the **JadeServer** section of the Jade initialization file.
- Assuming you are using multiple method caches, the method cache of the server thread is inherited from whatever had run there before - how likely is your method to be in that cache?

You could use a single method cache in the server node to avoid this, but that increases contention in the method cache, which in turn is likely to reduce overall performance.

- The server's object cache is used, rather than the node's. This could be a good thing or a bad thing.

If you are reading objects that are not in the node cache and would not be helpful in the node cache, it is a good thing.

If you access objects that were already in node cache, time is spent bringing them into the server cache as well.

If you had already updated the objects in the node cache in the same transaction, they have to be fetched from the client node.

- Accessing transients can be expensive if they are created on one node and used on another, as they must be copied back and forth. Does your server execution method contain references to **app**?
- Calling a client execution method from a server execution method is expensive, and it leaves the server thread in use. Passing transients back and forth becomes even more expensive.

The **serverExecution** method option is more useful than ever, but measure the difference it makes, if at all possible.

However, beware of the effects of micro-benchmarking (see "[Micro-Benchmarking – Measure as You Go](#)", earlier in this document). In addition, a server execution test will clearly run faster when no one else is using the database. It might be a lot slower during normal production, when the database is more-heavily loaded.

Deleting Transients

Although transient objects created by an application are removed when the application terminates, it is better to delete them as you go. Otherwise, you risk overloading your local JOM transient object cache with large numbers of unused objects. This will, in time, degrade the benefits of caching as your cache becomes full and has to be written to and retrieved from disk.

It is generally recommended that, unless you need to do so elsewhere, you delete your transients in the epilog of the method that created them. It is not sufficient to simply assume that the client will close down the application regularly enough to clear the orphaned transients. Note that there are occasions where it is appropriate to carry a transient for an extended duration, but it is important that it is deleted once it is no longer required.

Jade Care Start (**CardSchema**) provides a mechanism for monitoring undeleted transients at application shutdown. Jade Care Start is free to download from <https://www.jadeplatform.com/developer-centre/extensions>.

To enable this, set the **CheckTransientsOnShutdown** parameter to **true** in the [CardLog] section of your initialization file, or call **app.cnCheckForTransients**. It is recommended that you use these checks during application development and testing, but they should not be necessary in production.

If transient overflow is occurring, you can use the **Transient File Analysis** sheet in Jade Monitor to find out what objects are in the overflow file. You can use the **Transient File Summary** sheet to find processes with large overflow files.

Note To register multiple processes, you can multi-select on the **Users** sheet, or select one node at a time by right-clicking on the node line.

Jade Methods versus External Methods and Functions

In some circumstances, it may be more efficient to perform some processing in an external method or external function. A classic example of this is in string manipulation; for example, reading a large comma-separated values (CSV) file.

It is possible to develop a highly efficient text manipulation routine (for example, in C++) and to call that as an external method, more efficiently than the equivalent routine in Jade. This must be balanced against the development overhead of building and maintaining the external routine. The sort of circumstances where this may be of benefit is where large volumes of complex text manipulations or mathematical calculations are required. This is most-commonly encountered in bulk data load scenarios.

Very Large Methods, with Most Code Unused

A very large method, even if infrequently called, can effectively flush all other methods out of the method cache. It may pay to chop these up. Use the Jade User Interrupt Profiler to analyze method sizes. The issue here is not the run-time cost of the method, but the size of it and the subsequent effect on method caching; for example, there may be a method with a thousand lines of **if then** and **elseif then** statements. Only one of the conditions will ever be executed, but the code for all of them is being dragged into the method cache.

Ideally, the most-likely conditions would be checked first and the full checklist used only if required. The full checklist can be chopped up into several methods, with range checks to determine which method to call.

If the method is frequently called, ensure the chopping up doesn't cause too many method calls at run time (for details, see "[Method Calls](#)", in the following section).

Method Calls

There is an overhead in calling a method. In most cases, this is insignificant in relation to the overall processing of the system but in some circumstances a frequently used method that does very little can add a significant overhead.

Consider the following method on the **Integer** primitive type.

```
addOne(): Integer;
begin
    return self + 1;
end;
```

Consider also the following code fragment that uses this **addOne** method.

```
while int < 100000 do
    int := int.addOne;
endwhile;
```

In a brief test, this took 170 milliseconds to process the 100,000 iterations.

The following code fragment is a more-efficient approach.

```
while int < 100000 do
  int := int + 1;
endwhile;
```

In a brief test, this took 14 milliseconds to process the 100,000 iterations. This is much quicker because the overhead of calling the extra method is much greater than incrementing a local integer.

You can make this even more efficient by using the **foreach** instruction; for example:

```
foreach int in 1 to 100000 do
  .....
endforeach;
```

As well as being easier to use, **foreach** internally increments the value of the variable (in this case, **int**), meaning that it will always be faster than an equivalent **while** loop.

Fetching Properties

When your code accesses a property of an object, the property may have to be fetched from the object so it can be acted upon. Usually the processing time taken is inconsequential, but for large slob and blobs it can become significant. See also "[Slobs and Blobs](#)" in the "[IO Performance - Design Issues](#)" section of this document for a discussion about the IO characteristics of slob and blobs. For example, consider the following code, where **fileText** is a slob:

```
if self.fileText.length > 0 then
```

The **String::length** method runs very quickly, even if the slob is large, but fetching the property on which to run the method can be time-consuming.

In one production application, frequent use of code similar to the above accounted for 22 minutes out of a one-hour batch load process.

Excessive Logging

Routine logging can be invaluable for diagnostic purposes, but overuse can degrade performance. In one case, a single call to the **JadeLog::log** method from the wrong place increased a transaction from 3 seconds to 54 seconds.

If a lot of logging is required, use buffered logging with the **JadeLog::info** and **JadeLog::commitLog** methods.

Windows Calls – Setting Table Cell Text

If you are loading a lot of table cells, it is more efficient to load the row rather than each individual table cell. This is because the process of identifying each cell and then setting the **text** property requires a Windows call for each cell. Alternatively, if you put a tab-delimited string into the first cell of the row (or use the **addItem** method to load a tab-delimited string into the row), fewer Windows calls are required to insert each part of the string into the cells.

This was tested loading a table with 10 columns and 10,000 rows. Loading a tab-delimited string into the first cell of each row took just over half a second, compared with explicitly incrementing the **rows** property and loading each column individually, which took more than two seconds. Using the **addItem** method to populate the rows (and increment the **rows** property) took well under one second.

Multiple Concatenations

When a string is built up from a number of other strings, it is more efficient to do all of the concatenations in a single Jade statement; for example, consider the following code fragment.

```
displayName := parent.title;
displayName := displayName & " ";
displayName := displayName & parent.lastName;
displayName := displayName & ", ";
displayName := displayName & parent.firstName;
displayName := displayName & " ";
displayName := displayName & parent.middleName;
displayName := displayName & ", ";
displayName := displayName & parent.honorary;
```

In a test, the following code fragment produced the same result in seven microseconds compared to nine microseconds for the code fragment in the previous example.

```
displayName := parent.title
              & " " & parent.lastName
              & ", " & parent.firstName
              & " " & parent.middleName
              & ", " & parent.honorary;
```

Clearly for a savings of two-millionths of a second to become significant, you would have to execute this millions of times, so you won't see much performance benefit outside of repetitive string manipulation cases.

If possible, ensure all of the operands are literals or local variables, rather than object properties. The same test, when performed with local method variables, took 0.7 microseconds. Therefore if you are processing a flat file, for example, it would be better to use local variables rather than class attributes for work fields or to accumulate totals.

Although there are special optimizations in string concatenation for local variables, it is also generally true that method variables perform faster than object properties, which usually makes them more suitable for work fields.

System Methods or System Properties?

A lot of commonly used Jade constructs are properties, but there are a lot that are methods. Methods have to do some work to generate the return value, so they are sometimes significantly slower than properties.

Examples of commonly used methods are:

- **[Class.instances.size](#)**

The **[size](#)** method loops through the object index (part of the database infrastructure), counting the entries. Unlike Jade collections, which maintain a count in the header, the virtual collection returned in the **[Class::instances](#)** property does not have such a count.

The **[Class::countPersistentInstances](#)** method provides a faster way of obtaining the number of instances. You should use this method instead of the **[Class.instances.size](#)** method.

- **[Application::formsCount](#)**

This method counts the number of active forms.

- **[ListBox::listCount](#)**

This method counts the entries in a list box.

In many cases, the method can be called once and the result saved; for example, see "[Iterating Through Arrays](#)", earlier in this document, or the Jade Platform documentation for [ListBox::listCount](#) in the *Encyclopaedia of Classes (Volume 3)*.

beginTransaction and commitTransaction Instructions

When Jade processes a [beginTransaction](#) instruction, it must perform some processing to manage the transaction state. It must also ensure that any object that is locked during the transaction state remains locked until the [commitTransaction](#) or [abortTransaction](#) instruction point.

At the [commitTransaction](#) instruction point, Jade commits your updates to the database and releases all transaction-duration locks.

With most processing, the code executed in transaction state is determined by atomicity requirements but with bulk updates or data loads, this can be more flexible.

With bulk updates, you do not want to be jumping into and out of transaction state all of the time, as this carries an overhead that can become significant when repeated often enough. For example, each [commitTransaction](#) instruction requires that all of the database changes made within the transaction are written to the database audit journal. By grouping your updates together, you can reduce the overall amount of time spent waiting for writes to the journal.

Consider the following example code.

```
createCustomers();
vars
  i : Integer;
  cust : Customer;
begin
  beginTransaction;
  foreach i in 1 to 1000 do
    create cust;
    cust.id := i;
  endforeach;
  commitTransaction;
end;
```

Processing the 1,000 creates within a single transaction state (one [beginTransaction](#) and [commitTransaction](#) pair of instructions) takes about 145 milliseconds. Now consider the same code, but this time with a transaction state for each create, as shown in the following example.

```
createCustomers();
vars
  i : Integer;
  cust : Customer;
begin
  foreach i in 1 to 1000 do
    beginTransaction;
    create cust;
    cust.id := i;
    commitTransaction;
  endforeach;
end;
```

Processing the 1,000 creates with 1,000 transaction states (a [beginTransaction](#) and [commitTransaction](#) pair of instructions for each create) takes about 15,750 milliseconds.

By grouping the creates together within a single transaction state, you can greatly reduce the necessary overhead of locking and logging required to manage transaction states.

As with all processing approaches, a balance must be maintained. There may be a different balance between batch and online processing. For example, if you are creating hundreds of thousands of objects in a batch process, it would be detrimental to perform them all within a single transaction state if you also wanted to maintain online performance at the same time. You should therefore look to doing a [commitTransaction](#) and [beginTransaction](#) at some sort of regular interval, perhaps every 100 milliseconds, every few seconds, or whatever is most appropriate for your performance requirements.

See also "[Large Batch Processes](#)", later in this document.

IO Performance – Design Issues

This section provides IO performance design issue tips.

- [Slobs and Blobs](#)
- [Avoiding firstInstance and instances Methods](#)
- [Locking Strategy](#)
- [Using Local Collections](#)
- [Setting Key Values before References](#)
- [Map Files](#)

Slobs and Blobs

Embedded Strings, `StringUtf8s`, and `Binaries` come with the object when it is fetched from the database. `String Large Objects` (slobs), `StringUtf8 Large Objects` (slobutf8s), and `Binary Large Objects` (blobs) are fetched separately. Each fetch (Get Object) can result in one or more IOs to the disk.

Consider the following example.

```
Parent
  name          String[30]
  description   String[Maximum]
```

In this class, there will be one Get Object for each parent instance in this loop, as shown in the following code fragment.

```
foreach parent in myRoot.allParents do
  write parent.name;
endforeach;
```

There will be two Get Objects for each parent instance in this loop, as shown in the following code fragment.

```
foreach parent in myRoot.allParents do
  write parent.description;
endforeach;
```

There will be two Get Objects for each parent instance in this loop, as shown in the following code fragment.

```
foreach parent in myRoot.allParents do
  write parent.name;
```

```
        write parent.description;
    endforeach;
```

Jade creates slobbs or slobutf8s for **String** properties with a length of 539 or greater. Slobbs and slobutf8s are created for **Binary** and **StringUtf8** properties with a length greater than 540.

Slobbs, slobutf8s, and slobbs are also created if the length is specified as the maximum length.

Slobbs, slobutf8s, and slobbs are displayed in the Class Browser as having a **SubId** attribute, rather than being embedded.

Avoiding firstInstance and instances Methods

In general, it is best to avoid using the **instances** property and the **firstInstance**, **lastInstance**, **allSharedTransientInstances**, **firstSharedTransientInstance**, and **lastSharedTransientInstance** methods.

As these methods read a "dirty" OID list, you are potentially getting an inconsistent view of the data. For example, if you have done updates and not yet committed them, a straight database access will not reflect the uncommitted updates.

One valid use of this property and these methods in production systems is to retrieve a singleton persistent object, where there is only one instance of a class (for example, **Root**, as shown in the following code fragment). However, ensure that you save off a reference to the object rather than using the **firstInstance** method everywhere, because that requires a database access each time.

```
app.myRoot := Root.firstInstance;
```

For transient classes, the **instances** property and the **firstInstance**, **lastInstance**, and **instances** methods *do* provide a consistent view.

Locking Strategy

You should establish an effective locking strategy as part of your system design for every multiuser Jade application.

An effective locking strategy should meet the following general objectives, although in some circumstances these objectives can conflict. When this occurs, you should typically aim to maximize multithreading over other objectives.

- The locking strategy should support the greatest possible degree of multithreading (unless you are effectively in a dedicated single user mode; for example, where a solitary batch process is running offline).
- Objects should remain locked for the minimum time necessary.
- Objects (and classes) should be locked in a consistent order, to reduce deadlocks.
- Objects that are going to be updated should be locked as soon as that is known, with a **reserve** lock, **update** lock or **exclusive** lock.

There are discussions about locking (and deadlocking) issues throughout this document, and it is recommended that you review each of these discussions for a fuller understanding.

For more details about designing and building an effective locking strategy, read Chapter 6, "[Jade Locking](#)", in the *Developer's Reference*, and "Locking" in Part 4, "Design Considerations", in the *Erewhon Demonstration System Reference* included with the Example schemas and also available on the Jade website at <https://www.jadeplatform.com/developer-centre/learn/whitepapers>.

Using Local Collections

When designing your system, remember that an entire collection is exclusively locked whenever the collection is updated (for example, a new object is added to the collection, keys are changed, and so on), and that the lock will remain in place until the next `commitTransaction` (or `abortTransaction`) instruction.

The best thing you can do to manage this is to keep collection scope as local as possible. For example, when storing customer transactions, use a collection on the customer object rather than a global collection keyed by customer. This results in a separate collection instance per customer, and therefore a greatly reduced likelihood of contention.

When you need to have global collections, perhaps to facilitate reporting, then you should try to keep them locked for the shortest time possible. For example, you could use deferred updates or modify the keys or set the references immediately before the `commitTransaction` instruction. This will provide the greatest degree of concurrency.

Sometimes it is possible to offload the updating of global collections to a background process. For example, normal online users might create audit objects but not add them into the collection. A background process would then come along afterwards and add them to the collection. This can reduce the possibility of deadlocks, and avoid slowing down the online transaction.

Setting Key Values before References

When creating objects of a class, always set the attributes before you set the references. For example, consider the code fragment in the following example.

```
beginTransaction;  
  create sale;  
  sale.myProduct := product;  
  sale.name      := "widget"s;  
commitTransaction;
```

When `sale.myProduct` is set, the sale object is added into a collection on `Product`, which has the key `Sale::name`. As `sale.name` has not yet been set, the sale is added with a key value of null. When `sale.name` is subsequently set, Jade will have to go back and change the collection because the key has now changed.

The code fragment in the following example is considerably more efficient, as Jade will only have to update the collection once.

```
beginTransaction;  
  create sale;  
  sale.name      := "widgets";  
  sale.myProduct := product;  
commitTransaction;
```

In a test with 50,000 sales, the first code fragment example took over 34 seconds, while the second code fragment example took just over 6 seconds.

Map Files

Map files store the physical contents of your database. The term *map file* is a contraction of *class map* and *database file*. The schema and physical entity to which it refers is a database file.

In general terms, you should partition your database over multiple map files, to reduce the overhead of managing large map files. This can become particularly important when reorganizing your database.

Some general rules to apply to map files are as follows.

- Ensure that your largest (the most objects and greatest data) classes are each in a separate map file.
- Do not put small classes (with a low number of objects) into the same map file as a large class (with a large number of objects). This is because, in the case of a reorganization, the whole map file must be reorganized so a reorganization of the small class is a lot slower because of the large map file.
- Lumping a number of small classes together in a map file is a good idea as it reduces the number of map files, but be sure to consider database growth.
- Consider using database file partitioning so that objects of a class can be spread over multiple files.
- Well-designed databases normally have between 50 and 200 map files, depending on their size and the complexity of the data model.
- Consider locating map files over multiple separate physical devices, to spread the load over multiple physical IO devices. Normally you would use RAID disk to accomplish this at the hardware level.
- In a production system, database journals should be located on a separate physical device from the map files. This is both for performance reasons and for recoverability in the event of media failure.
- Use the [DefInitialFileSize](#) and [DefFileGrowthIncrement](#) parameters in the [\[PersistentDb\]](#) section of the Jade initialization file to help reduce disk fragmentation. The values in the following example are a good starting point for many databases.

```
[PersistentDb]
DefInitialFileSize=100K
DefFileGrowthIncrement=1M
```

Using these values, the map file will initially be created at 100K bytes. When it grows beyond that size, it will be extended to 1M byte, 2M bytes, and so on. Each time the file grows, there is the possibility that disk fragmentation may occur, so using a smaller increment will cause more fragmentation. Using overly large values will result in a lot of disk being allocated but not yet used. Generally, the larger the file is expected to be, the larger the value of the [DefFileGrowthIncrement](#) parameter should be.

The settings in the Jade initialization file apply to all map files. You can set these parameters for individual map files by using the Jade Database utility. You can also use the Jade Database utility to preallocate an entire file, if its size is known ahead of time.

IO Performance – Development Issues

This section provides IO performance development issue tips.

- [Iterator versus the foreach Instruction](#)
- [Minimizing the Working Set](#)
- [Jade Implicit Locking](#)
- [Releasing Locks](#)
- [Large Batch Processes \(Adds, Deletes, or Updates\)](#)
- [Reducing Deadlocks](#)
- [Defining a Locking Strategy](#)

Iterator versus the foreach Instruction

The two main techniques for retrieving a number of objects from a collection are as follows.

- Iterator
- **foreach** instruction

There is a lot of confusion about these, and the impact the different constructs have.

Usage

The following methods are examples of the typical usage of an iterator and the **foreach** instruction. It is the simplest case, where all objects in the collection are to be accessed.

```
usingIterator();
vars
    parent : Parent;
    iter : Iterator;
begin
    iter := global.myRoot.allParentsByName.createIterator;
    while iter.next(parent) do
        // do something with parent
    endwhile;
epilog
    delete iter;
end;

usingForeach();
vars
    parent : Parent;
begin
    foreach parent in global.myRoot.allParentsByName do
        // do something with parent
    endforeach;
end;
```

Both constructs will go sequentially through the collection and return each object reference in turn. Both support the **break** and **continue** instructions. The **foreach** instruction is a little simpler. The iterator has features that the **foreach** instruction does not.

Locking Behavior

In the case of the iterator, the first time **iter.next** is executed a group of object references will be fetched from the database (object references are also known as *object IDs*, or OIDs). The size of the group is normally 51, but it is less for dictionaries with large keys.

The collection is implicitly share locked for the duration of the fetching of the group of OIDs. The lock is released when the group of OIDs is sent to the requesting node. When the group of OIDs is exhausted, another group will be fetched from the database. The collection is locked and unlocked each time another group is fetched.

The **foreach** instruction implicitly share locks the collection at the start of the iteration and releases the lock when the **endforeach** instruction is reached.

If the **discreteLock** option is used with the **foreach** instruction, the locking aspect of the construct is the same as the iterator; that is, the collection is locked each time a group of OIDs is fetched and it is released when the OIDs are returned.

If the iterator construct is surrounded by **beginLoad** and **endLoad** or **beginTransaction** and **commitTransaction** instruction pairs, the locking behavior becomes like that of the **foreach** instruction without the **discreteLock** option. This is because when in transaction state or load state, all transaction duration locks are held until the next **endLoad**, **commitTransaction**, or **abortTransaction** instruction.

Note The **Dictionary** class **getIteratorKeys** method share locks the collection on each call. If you are in load state or transaction state, the locking may not matter. The **Iterator** class **getCurrentKey** and **getCurrentKeys** methods can be used as an alternative to avoid the locking overhead.

Starting Part-Way through the Collection

One of the key benefits of the iterator is the ability to start efficiently at any point in the collection and to scan forward or backward from there. For example, you can read through a portion of the collection, as shown in the following example of a range of parent id numbers.

```
usingIterator();
vars
  parent : Parent;
  iter : Iterator;
begin
  iter := global.myRoot.allParentsById.createIterator;
  global.myRoot.allParentsById.startKeyGeq(id, iter);
  while iter.next(parent) do
    if parent.id > target then
      break;
    endif;
    // do something with parent
  endwhile;
epilog
  delete iter;
end;
```

The optimization shown in the preceding example is available with the **foreach** instruction. In this example, the start key test is **>=** and the key is not declared descending.

```
usingForeach();
vars
  parent:Parent;
begin
  foreach parent in global.myRoot.allParentsById
  where parent.id >= id do
    if parent.id > target then
      break;
    endif;
    // do something with parent
  endforeach;
end;
```

For full details about this optimization, see the *Developer's Reference*.

Notes The **foreach** instruction optimization is for simple key tests on single-key dictionaries only. If the key is a `TimeStamp` called **contractStart**, the **where contractStart.date >= now** test is not a simple key test. Each object must be fetched to run the **date** method on the **contractStart** attribute to derive the key test.

For any other start condition or if you are not sure if your **foreach** instruction will be optimized, use an iterator and one of the **startKey** methods.

Don't forget to break out of the loop when you have finished; for example, when populating the visible portion of a combo box or table. Fetching and ignoring the rest of the objects in the collection can waste a lot of time!

Using Collection Keys to Avoid Fetching Objects

When you are iterating through a collection and the information you require is contained on the keys of the collection, you may not need to fetch the objects referenced by the collection.

Remember that the **foreach** instruction and **iter.next** return object ids (OIDs); they do *not* fetch the objects. The objects are not fetched until an attribute is referenced, or a method is executed on the object. For example, if you have a collection of invoices stored by date and the only information you actually want is the invoice date, you do not need to access the actual objects of the **Invoice** class referenced by the collection. In this case, you can use **Iterator::getCurrentKeys** or **Iterator::getCurrentKey** to retrieve the key values without having to fetch the actual invoices.

Notes The benefit in a live production system of saving the fetch of the object is normally IO time. If you do some micro-benchmarking in this area, beware of the effects of caching. To clear the caches, restart the database server, then run the test.

The **Iterator::getCurrentKey** method locks the collection. If you are in load state or transaction state, the locking may not matter. (The **Iterator::getCurrentKeys** method was introduced in Jade 6.2.14 as an alternative to the **Dictionary::getIteratorKeys** and **Iterator::getCurrentKey** methods, to avoid share locking the associated dictionary on each call.)

This technique is useful when you:

- Need one or more key values but you don't need any other attribute of the object.
- Can avoid fetching most of the objects by checking the keys first (filtering).

This technique is not useful when:

- You will be fetching the objects anyway.
- The keys you need to check are in a collection but not the collection over which you are iterating.

In some cases, it may be beneficial to add low-order keys to collections to facilitate filtering.

Minimizing the Working Set

In loops where there are multiple filters, apply the cheapest filters first and then the filters that reduce the working set the most. For example, consider the following code fragment, which finds sales of appliances in a specified city.

```
while iter.next(tran) do
  if tran.type = Type_Sale
    and tran.myBranch.myLocation.city = targetCity
    and tran.myProduct.isAppliance then
      <do something with tran>
```

```
    endif;  
endwhile;
```

In this example, **tran.type** should be checked first, because it is the cheapest. The **tran** object must be fetched to evaluate all of the other conditions, so we may as well check the **type** attribute first. If we did the **isAppliance** check first, we would have to fetch all of the product objects for the transactions that were not sales. Regardless of how many transactions are sales and how many products are appliances, it will save time to check **tran.type** first.

Now, assume that:

- 80 percent of transactions are sales
- 15 percent, on average, are likely to be in the target city
- 90 percent of the products are appliances

It pays to check the city first, even though it means fetching the branch and location objects for the non-appliance products. There are very few non-appliance products, so the number of extra fetches is small. By contrast, checking for non-appliance products for all other cities would result in a large number of extra fetches.

It doesn't matter if the filters are conditions of an **if** instruction, multiple **if** instructions, or multiple conditions in the **where** clause of a **while** statement; the end result is the same.

This code fragment example is simple and concise, to convey the concept. In the real world, each successive filter may be in another method, another class, or even another schema. It may take a bit of investigation to find all of the filters involved in a single loop.

Jade Implicit Locking

Jade puts a shared lock on collections when reading them, to prevent them being altered by another process. This lock is released when the read function is finished, unless the process remains in either transaction or load state.

Jade will automatically lock an object (including a collection object) if that object is updated in any way. Normally, this will be an exclusive lock, but it will be an update lock if the updating process has enabled update locks, and the object being updated is a collection. Jade does this implicit locking to ensure that:

- No one else will view that object in an incomplete state.
- No one else will view uncommitted data, as that update may be backed out if the transaction aborts.
- You have the latest edition of an object in cache before you update it. (Jade has an underlying principle that it will never allow you to update an out-of-date object; that is, dirty write.)

Releasing Locks

Remember that as part of **commitTransaction** or **abortTransaction** instruction, Jade will release all transaction duration locks. This includes all types of locks; that is:

- Share
- Reserve
- Update
- Exclusive

It does not matter if the object was updated or not, or how the lock was obtained; that is:

- Implicitly locked automatically by Jade
- Explicitly when your code included a lock instruction, or similar

It also does not matter when the lock was obtained; that is:

- Before the `beginTransaction` instruction
- Within transaction state

Session duration locks are not released at the end of transactions or at the end of load state. For more details, including locking and unlocking in load state and lock state, see Chapter 6, "[Jade Locking](#)", in the *Developer's Reference*.

Large Batch Processes (Adds, Deletes, or Updates)

The suggestions in this section apply whenever you are doing a large number of adds, deletes, or updates.

Consider running the batch process as server execution methods, a server application, or running in single user mode. All of these options will generally improve performance. For potential impacts on other users, see "[serverExecution Method Option](#)", earlier in this document.

If running in multiuser mode, consider breaking up your transactions so that updates to a centralized collection, which multiple processes may want to access, are separated from the main bulk updates. This may help to reduce single-threading when Jade does its exclusive lock on the updated collection. In a situation where there is one updater and one or more readers, using update locks can also help reduce single-threading.

For example, in the following code fragment, as soon as a new company is discovered, it will be created and added to the collection off `Root`. Jade will put an exclusive lock on the collection `root.allCompanies`, which means that no one else will be able to read this collection until this process finishes and performs the `commitTransaction` instruction to unlock everything that has been locked.

```
beginTransaction;
foreach record in company sales input file do
    company:= app.myRoot.allCompanies.getAtKey(record.companyID);
    if company = null then
        create company;
        company.ID := record.companyID;
        company.name := record.companyName;
        company.myRoot := app.myRoot;
    endif;
    create sale;
    sale.date := record.date;
    sale.myCompany:= company;
endforeach;
commitTransaction;
```

A better approach may be to read through the input file first, just to establish if any new companies needed adding and add them at that stage. You would then re-process the file and perform the sale update process.

You could also do a `commitTransaction` every 1,000-10,000 objects or after a specific interval. This can help keep the transactions from becoming too large and to allow other processing to gain access to locked resources.

The best number of objects per transaction can be difficult to determine, so it may be easier to simply commit based on elapsed time, perhaps every 500 milliseconds or every 2 seconds. You should base the specified time on how long locks can be held without introducing problems, and how large the transaction is. Overly large transactions can cause additional resource consumption.

Another advantage of time-based commits over count-based commits is that they automatically reduce the impact on other users during peak periods. You may process 1,000 objects in 0.5 seconds during idle times, but it may take 6 seconds during peak times, resulting in locks being held longer when you can least afford them. Time-based commits automatically include fewer objects in each commit during peak times.

Breaking a single large update into multiple smaller ones can help performance, but it can also introduce recovery and concurrency issues. If the server goes down, you need to be able to restart and reprocess from the point to which the database recovered. If other users can access the objects during the load, they need to be committed in a coherent state.

Reducing Deadlocks

An example of a deadlock situation is where you have a process (for example, **P1**) that locks an object (for example, **Ob1**), and another process (for example, **P2**) that locks an object (for example, **Ob2**). If **P1** now attempts to lock **Ob2** and **P2** attempts to lock **Ob1**, both processes will be blocked from completing by the other process.

As soon as Jade identifies a deadlock situation, the process that triggered the deadlock will be given a deadlock exception and the operation will be aborted. In the above example deadlock situation, Jade would give the deadlock exception to **P2**.

Note For a deadlock situation to occur, at least two of the locks involved must be **reserve**, **update**, or **exclusive** locks. If all of the locks are **share** locks, there will be no deadlock.

As another example of a deadlock situation, if two processes have **share** locks on the same object and they both try to upgrade the lock to **reserve** or **exclusive**, a deadlock will occur. In this case, the second process to attempt the upgrade receives the deadlock exception. If two processes have **share** locks on the same object and they both try to upgrade the lock to **update**, a deadlock will not occur. This is because the **share** lock is released before the **update** lock is requested, even if in transaction state.

The more deadlocks you have occurring in your system, the greater the impact will be on the performance of the system, let alone the irritation to the users. It is therefore sensible to take practical steps to reduce the number of deadlocks that occur. There are a number of simple steps to take to assist in this; for example:

- Control the order in which classes are locked. For example, always lock the customer first, then lock the customer's accounts, then lock the customer account's transactions.

This is probably best enforced by encapsulating the locking activity.

- Within a class, control the order in which objects are locked. For example, when locking multiple customers, always try to lock in one specific order (for example, the customer number order).

Enforce this by encapsulating the locking activity.

- Avoid using the **beginLock** and **endLock** instruction pair, as it potentially locks too many objects and can make identification of deadlock causes very difficult.

If you impose these sorts of disciplines on all processes, instead of getting deadlocks you will simply wait for locked resources to become available. This is preferable for the end-user and removes the cost of backing out aborted transactions. It is a lot easier in a project to impose these sorts of disciplines if the operations involving locking and updating are centralized in an application.

You can use the `Process::setPersistentDeadlockPriority` method to control which user will receive the deadlock exception. You could give higher priority to online users, for example, so that background reports receive the deadlocks or give higher priority to the background reports so that they don't hold locks as long.

If all processes involved in a deadlock have the same deadlock priority, the exception is usually given to the process whose action caused the deadlock.

You can set the `DoubleDeadlockException` parameter to `true` in the `[JadeServer]` section of the Jade initialization file to help diagnose deadlocks. With this setting, both processes involved receive the deadlock exception, which enables you to get two stack dumps to diagnose the contention.

Defining a Locking Strategy

A good strategy for controlling locking is to use encapsulation to control access to properties so that an object has control of its own destiny (and that of its related subobjects).

In general, make attributes `readOnly` in persistent classes, references protected, and try to avoid set methods on references, as this will reduce the amount of control you have in enforcing a locking strategy.

For more details about designing and building an effective locking strategy, read "Locking" in Part 4, "Design Considerations", in the *Erewhon Demonstration System Reference*.